

Trabalho prático individual nº 2

Representação do conhecimento

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2019/2020

29-30 de Novembro de 2019

I Important remarks

1. This assignment is to be completed individually by each student, within 27 hours after this description is published. Assignments can be submitted beyond the 27 hours, with a penalty of 5% per each additional hour.
2. Submit the requested classes and/or functions in a single module named "tpi2.py" and include your name and number; don't submit other modules.
3. You can discuss the assignment with your colleagues, but you cannot copy the requested code from any source.
4. If you discuss the assignment with other colleagues, identify them in the modules with name and number. If you use any other sources, identify them as well.
5. All submitted code should be original; although trusting that most students will submit their own work, code plagiarism will be checked with appropriate tools. Students caught cheating will automatically fail the assignment.
6. Programs will be evaluated based on: correctness and completeness (70%); code style (10%); and originality / evidence of independent work (20%). Correctness and completeness will be primarily based on automated testing, but if your work is not fully functioning, it will be analyzed by the course staff to give due credit.

II Exercices

Attached to this description, you can find the `semantic_network` and `bayes_net` modules, which you are not allowed to change. These modules are similar to those used in practical classes, with small changes. You are requested to solve the proposed exercises exclusively in the module `tpi2`. Also attached to this description, you can find the module `tpi2_tests`, which contains several tests for the requested functions. If needed, you can add other test code in this module.

1. In applications involving causal dependency relations, as is the case of diagnosis applications, it makes sense to give special treatment to these dependencies. Thus, the **Depends** class, derived from **Relation**, was added to the `semantic_network` module. As an example, and to support testing, you can find in the module `tpi2.tests` a semantic network to support inferences about malfunctions in cars. This network is partially shown in Figure 1 (for simplicity, `debug_time` associations are omitted in the figure, but you can consult them in the module).

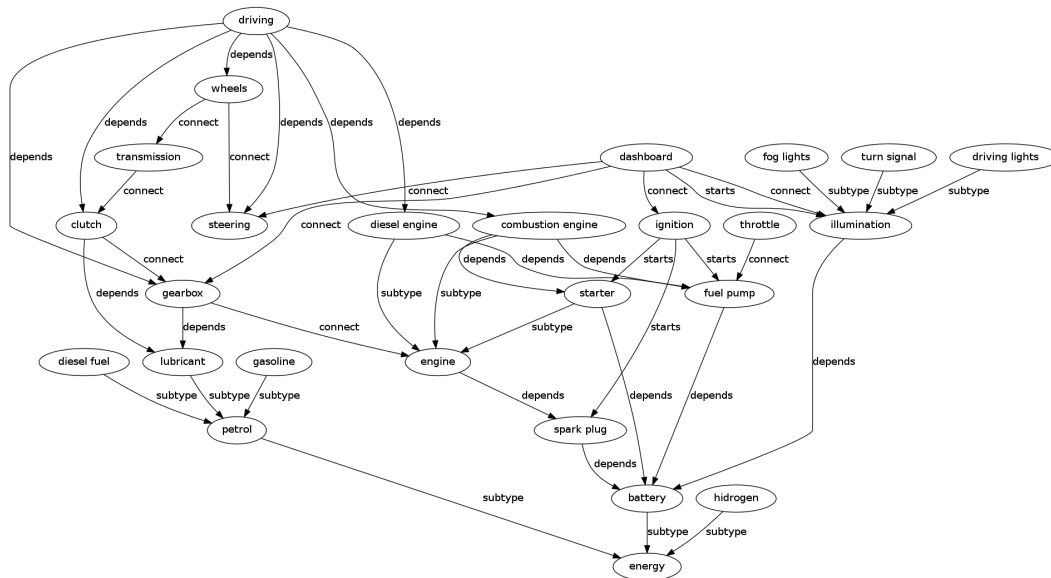


Figure 1: Main relations in the car domain.

- a) Develop a method `query_dependents()` in the class `MySN` that, given an entity (typically a type of car component), E , returns the list of all entities whose operation depends on the correct operation of E . These dependencies are given by the **Depends** relation and can also result from chaining several such relations. This way, 'engine' depends on 'battery' through 'spark plug'. Moreover, these dependencies are inherited by subtypes. For example, 'driving lights' is a subtype of 'illumination', therefore it depends on 'battery'. The list that will be produced by the requested method should not include supertypes (i.e. entities with subtypes). For example, one of the dependents of battery is 'driving lights', because it is a subtype of 'illumination', so the latter is not included. Examples:

```
>>> z.query_dependents('battery')
['turn signal', 'fuel pump', 'driving', 'fog lights',
 'driving lights', 'diesel engine', 'combustion engine',
 'spark plug', 'starter']
>>> z.query_dependents('illumination')
[]
>>> z.query_dependents('engine')
['combustion engine', 'driving']
```

- b) In such applications, it is especially important to identify the potential causes of a malfunction. Develop a method `query-causes()` in the class `MySN` that, given an entity (component, functionality), E , returns the list of all entities whose failure or malfunction can cause failure or malfunction in E . The previous function traverses the chain of dependencies from causes to effects, whereas the function `query-causes()` does the opposite. Again, types having subtypes should not be included.

Example:

```
>>> z.query-causes('driving')
['fuel pump', 'battery', 'clutch', 'diesel engine', 'combustion engine',
 'spark plug', 'lubricant', 'wheels', 'gearbox', 'steering', 'starter']
```

- c) If the car breaks down, the causes of the problem must be checked. Each component takes some time to be analyzed, so it is useful to sort potential causes by increasing order of the required analysis time. Develop a method `query-causes_sorted()` in the class `MySN`, which, given an entity (component, functionality), E , returns a list of tuples (X, T) , where X is an entity whose failure or malfunction can cause damage or malfunction of E (X is a potential cause of the problem observed in E) and T is the time required to analyze X . The function returns these tuples for all potential causes as a list sorted in ascending order of time. For the identification of the causes, you can use the function from the previous exercise. As each technician takes a different amount of time, the sorting should be based on average time (see information on module `tpi2.tests`).

Example:

```
>>> z.query-causes_sorted('driving')
[('wheels', 1), ('battery', 10), ('lubricant', 16), ('fuel pump', 60),
 ('starter', 65), ('steering', 65), ('spark plug', 137), ('clutch', 190),
 ('gearbox', 203), ('combustion engine', 245), ('diesel engine', 270)]
```

2. Bayesian networks have several mathematical properties. One such property is the *Markov blanket*. It is defined as a set containing all the variables that protect a given variable from the rest of the network. All knowledge about this variable can be extracted from the knowledge of its Markov blanket. The Markov blanket of a variable A in a Bayesian network is the set of nodes ∂A consisting of the parents and children of A and the other parents of the children (see Fig. 2).

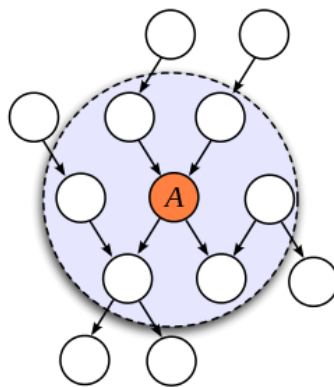


Figura 2: Markov blanket of a variable A .

Implement in the MyBN class a `markov_blanket()` method that, given a variable of the network, `var`, returns a list of the variables that make up the Markov blanket of that variable.

For testing purposes, the network represented in Figure 3 is included in the `tpi2_tests` module.

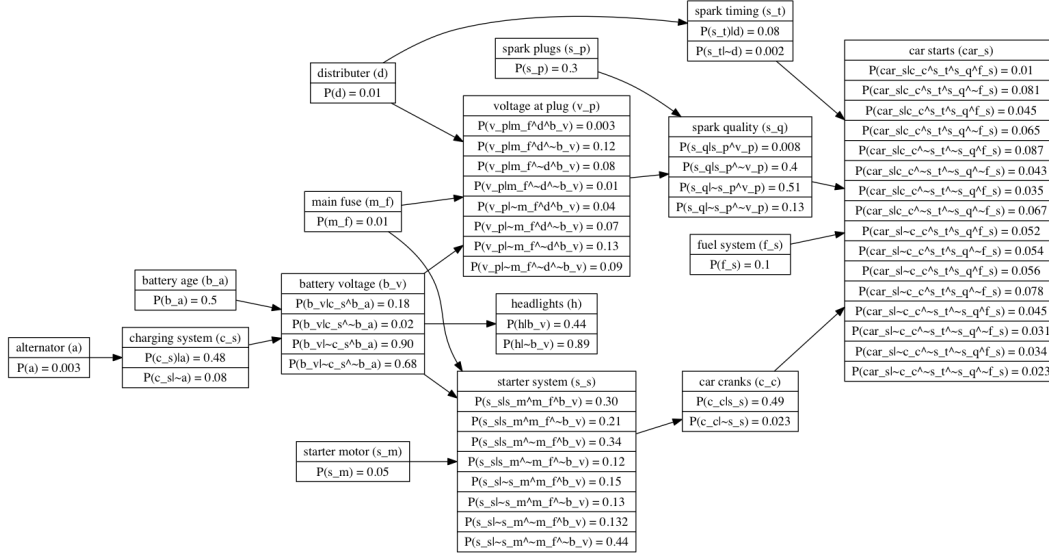


Figure 3: Bayesian network for a car that does not start

Examples:

```
>>> markov_blanket('s_t')
['d', 'car_s', 'f_s', 's_q', 'c_c']
>>> markov_blanket('c_s')
['a', 'b_v', 'b_a']
```

III Clarification of doubts

This work will be followed through <http://detiuaveiro.slack.com>. The clarification of the main doubts will be placed here.

1. Can we add new arguments to the functions?

Answer: You can add optional arguments only. When testing, the function call will contain only the arguments that were specified in the assignment description.

2. In 1.c), when we have causes with the same average time, is there a second criterion to break the tie?

Answer: It is a good idea to have the second criterion based on the alphabetical order of the cause; this way, ('starter', 65) should appear before ('steering', 65); the example was corrected above.

3. What is the weight of each exercise for marking?

Answer: The four exercises will have equal weight.

4. In 1.a) and 1.b), the lists must be sorted as in the examples?

Answer: No. Only in 1.c) the order is important.

5. "set of nodes ∂A consisting of the parents of A , their children, and the other parents of their children": "their children" is referring to the children of A, not to children of the parents of A, right?

Answer: Yes, you are right. That definition was rephrased (see above).

6. Shouldn't the result of `z.query_dependents('engine')` also include the '**starter**' and '**diesel engine**'? They are subtypes, not supertypes.

Answer: '**starter**' and '**diesel engine**' are subtypes of '**engine**', not dependents.

7. Could you provide more examples for 1.b)?

Answer:

```
>>> z.query-causes('fog lights')
[ 'battery' ]
>>> z.query-causes('gasoline')
[ ]
```